A Memetic Algorithm with a Large Neighborhood Crossover Operator for the Generalized Traveling Salesman Problem

Boris Bontoux*,[a], Christian Artigues[b], Dominique Feillet[c]

[a] *Université d'Avignon et des Pays de Vaucluse,*
Laboratoire d'Informatique d'Avignon
F-84911 Avignon Cedex 9, France
[b] *LAAS CNRS, Université de Toulouse*
7 avenue du Colonel Roche, 31077 Toulouse CEDEX 4
[c] *Ecole des Mines de Saint-Etienne, CMP Georges Charpak,*
F-13541, Gardanne, France

Abstract

The Generalized Traveling Salesman Problem (GTSP) is a generalization of the well-known Traveling Salesman Problem (TSP), in which the set of cities is divided into mutually exclusive clusters. The objective of the GTSP consists in visiting each cluster exactly once in a tour, while minimizing the sum of the routing costs. This paper addresses the solution of the GTSP using a Memetic Algorithm. The originality of our approach rests on the crossover procedure that uses a large neighborhood search. This algorithm is compared with other algorithms on a set of 54 standard test problems with up to 217 clusters and 1084 cities. Results demonstrate the efficiency of our algorithm in both solution quality and computation time.

Key words: Genetic Algorithm, Traveling Salesman Problem, Large Neighborhood Search

*Corresponding author
Email addresses: boris.bontoux@univ-avignon.fr (Boris Bontoux), artigues@laas.fr (Christian Artigues), feillet@emse.fr (Dominique Feillet)

1. Introduction

In this paper, we propose a solution method for the Generalized Traveling Salesman Problem (GTSP) based on a Memetic Algorithm (Genetic Algorithm plus Local Search, see [1] or [2] for further details). The GTSP is a generalization of the well-known Traveling Salesman Problem (TSP). The main contribution of the paper stands in the crossover operator based on the exploration of a large neighborhood around the father and mother individuals.

The GTSP can be described as follows. Let $G = (V, E)$ be a complete undirected graph, $V = \{v_1, \ldots, v_n\}$ a set of cities and $W = \{W_1, \ldots, W_m\}$ a set of clusters, with $0 < m \leq n$. Each city $v_i \in V$ belongs to exactly one cluster (note that from this definition the clusters are mutually disjoint). Let c_{ij} denote the routing costs for $v_i, v_j \in V$. The objective is to find a tour visiting exactly once each cluster while minimizing the sum of the routing costs. In this work, we only consider symmetric cost matrices ($c_{ij} = c_{ji}$), but the algorithm could easily be generalized to the asymmetric case. In particular, the crossover operator can be indifferently applied on symmetric or asymmetric instances.

The GTSP is NP-hard in the strong sense since it generalizes the TSP. Indeed, the special case where $m = n$ (a city per cluster) is a TSP: the problem resorts to find a tour visiting each city at a minimum cost.

In Section 2, we review the literature on the GTSP. Section 3 presents a new Memetic Algorithm developed for the GTSP. The main characteristic of this algorithm is its Large Neighborhood Search crossover procedure (see [3] for a recent work on Large Neighborhood Search techniques). Section 4 provides a computational evaluation of our algorithm through benchmark instances from the GTSPLIB [4].

2. State of the art

The GTSP was first introduced by Srivastava *et al.* [5] and Henry-Labordere [6], each one proposing to solve it through dynamic programming. Laporte and Norbert [7] and Laporte *et al.* [8] developed integer programming formulation,

permitting to solve exactly the GTSP with Branch & Bound techniques. More recently, an efficient Branch & Cut solution scheme was proposed by Fischetti *et al.* [9], who provide optimal solutions for instances with up to 89 clusters and 442 cities.

Many attempts have been done to transform efficiently the GTSP into the TSP [10, 11, 12, 13, 14]. Some of the resulting TSP instances have nearly the same number of nodes as the original GTSP instances. Moreover, some transformations of the GTSP into the TSP [11] have an important property: an optimal solution to the related TSP can be converted to an optimal solution to the GTSP. Unfortunately, a feasible non-optimal solution for the TSP may be not feasible for the GTSP. Furthermore, well-known heuristics for the TSP may not perform well for the GTSP.

Two approximation algorithms have been published for the GTSP. Slavík [15] presented a $3\rho/2$-approximation algorithm for the GTSP, where ρ is the number of cities in the largest cluster ($\rho = \max_{i=1,\ldots,m}(\mid W_i \mid)$). Unfortunately, the worst-case bound may be relatively weak, as ρ may be quite large. Garg *et al.* [16] proposed an approximation algorithm for the group Steiner tree problem, which provided an $O(\log^2(n)\log(\log(n))\log(m))$-approximation algorithm for the GTSP. In both cases, the triangle inequality must be satisfied.

In [17], Noon proposed several heuristics, including an adaptation of the nearest-neighbor heuristic developed for the TSP. Similar adaptations have been implemented by Fischetti *et al.* [9], such as farthest-insertion, nearest-insertion and cheapest-insertion. More recently, Renaud and Boctor [18] proposed an heuristic called GI^3 (Generalized Initialization, Insertion and Improvement), which is a generalization of the I^3 heuristic presented in [19] for the TSP. This heuristic consists of three phases: an initialization during which a partial tour is constructed, an insertion phase which completes the tour by inserting at the cheapest cost cities from unvisited clusters and an improvement phase based on 2-opt and 3-opt moves between clusters, called here G2-opt and G3-opt. The authors also explain how the cheapest sequence of cities visiting the set of clusters in a given order can be determined in polynomial time. They present a

procedure called ST algorithm (for Shortest Tour).

Snyder and Daskin [20] proposed to solve the GTSP with a Genetic Algorithm using a random-key encoding which assures that solutions constructed by crossover or mutations are feasible. The Genetic Algorithm was coupled with local search improvement, namely a swap procedure and a 2-opt neighborhood search, yielding a Memetic Algorithm. Computational results show the efficiency of their algorithm, in terms of solution quality and computation time. A Particle Swarm Optimization based algorithm was also recently developed by Shi *et al.* [21].

Finally, Silberholz and Golden [22] very recently proposed a Genetic Algorithm with several new features, including isolated initial populations and a new reproduction mechanism, based upon the TSP ordered crossover operator. This new mechanism was called *mrOX*, for modified rotational ordered crossover. Local improvement procedures combined with this mechanism, yielding again a Memetic Algorithm, permit to obtain very good results on large new instances. This algorithm can be considered as the most competitive algorithm published to date.

3. A new Memetic Algorithm

A Genetic Algorithm is a search technique widely used to find approximate solutions of optimization problems (see, *e.g.*, [23, 24, 25]). Genetic Algorithms are categorized as metaheuristics and are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover. Genetic Algorithms maintain a large number of solutions during the solution process. The set of solutions is called the *population*. Each solution is called an *individual*. At each iteration of a Genetic Algorithm, a new population is generated using several operators: reproduction, crossover and mutation.

Genetic algorithms paired with local search techniques are categorized as Memetic Algorithms [2, 26, 1]. In this section, we present a new Memetic Algorithm. We particularly insist on the crossover operator, which is our main

contribution. Moreover, to clearly evaluate the impact of this operator, we voluntarily adopt a very standard implementation for the rest of the algorithm.

3.1. Basic components of the algorithm

3.1.1. Individuals

Each individual (a solution of the problem) is represented by an ordered list of clusters, where the first and last clusters are identical. From this representation, a city tour can be derived, defined as the optimal tour maintaining the visiting order of clusters. The cost of the individual is the cost of this city tour. It is obtained using the Shortest Tour algorithm developed by Renaud and Boctor [18].

The principle of the Shortest Tour algorithm is the following. A succession of clusters defines a sequence of sets of cities, where cities from one cluster can only be attained from cities belonging to the preceding cluster. Representing cities by nodes, we obtain a directed acyclic graph. In this graph, the subset of paths having identical starting and ending nodes exactly corresponds to the set of GTSP solutions respecting the order defined by the cluster sequence. The best of these solutions coincides with the shortest path of this subset. Seeing that calculating the shortest path in an acyclic graph can be done in polynomial time with a simple recursion, the best city tour can easily be obtained. One just has to successively consider each city of the first (and last) cluster and compute the shortest path constrained to start and end with this city. The optimal city tour is the best path found during this process. The Shortest Tour procedure is computationally cheap and, consequently, can be called very often (more precisely, its complexity is $O(n^3/m^3)$ – see [18]).

3.1.2. Initial population

Our initial population contains N individuals. Individuals are constructed through randomly generated cluster lists. The Shortest Tour algorithm is applied to determine the optimal city tour and the cost for each individual. In order to avoid symmetries, the first (and last) cluster is identical for all individuals; in order to limit the computation time of the Shortest Tour procedure,

the cluster containing the fewest cities is chosen.

3.1.3. Population renewal

At each generation, two individuals are randomly chosen through Roulette Wheel Selection and paired for crossover. These two parents breed two children. This operation is repeated k times. The children are then added to the population and only the N best individuals are kept. A mutation procedure is applied to improve the population diversity and avoid premature convergence. Each individual has a probability μ of being selected for mutation (in our experiments $\mu = 0.05$). The mutation consists in swapping two randomly chosen clusters and applying the Shortest Tour algorithm to compute the optimal city tour and the new individual cost. The heuristic stops when N_1 generations have been computed or when no improvement has been performed during N_2 generations.

3.1.4. Memetic Algorithm

The proposed Memetic Algorithm associates all the elements described above, plus the crossover operator and the local search procedures presented in the next sections. Figure 1 presents a synthetic view of the algorithm.

Algorithm 1 Memetic Algorithm

```
Compute an initial population of N random individuals
Apply local search on these individuals
while the number of iterations is lower than N1 and an improvement has
occurred since less than N2 iterations do
  for i = 1 to k do
    Choose 2 individuals randomly
    Construct 2 children with crossover
    Apply local search on both children
    Add children to the population
  end for
  Keep the N best individuals in the population
  Apply mutation with a probability μ to every individual of the population.
end while
```

3.2. Crossover operator

The crossover operator is very important in a Genetic or a Memetic Algorithm. This operator allows constructing new solutions from existing solutions

and plays a great part in the behavior of the algorithm.

A crossover, or reproduction, is the equivalent of two parents mating and producing two children. These children bear a resemblance to each parent. Several crossover operators have been proposed for the TSP or for TSP-like problems: *e.g.*, the maximal preservative crossover (MPX, [27]) or the modified rotational ordered crossover (mrOX) proposed by Silberholz and Golden [22]. A comparison of different crossovers used for the TSP is presented in [28].

The crossover procedure we propose here is inspired from the *dropstar* procedure used in Bontoux and Feillet [29]; shortly, in that paper, the context was the solution of the Traveling Purchaser Problem and *dropstar* was used as a local search operator determining the best subsequence from a sequence of cities. It is noteworthy saying that this operator is also inspired from the algorithm used by Prins in [30].

Let $W_{i'_1}, \ldots, W_{i'_m}$ and $v_{i_1}, \ldots, v_{i_m}$ respectively be the cluster tour and the derived city tour of an individual, called the father. Let $W_{j'_1}, \ldots, W_{j'_m}$ and $v_{j_1}, \ldots, v_{j_m}$ respectively be these tours for another individual, called the mother. A new individual - a child - is built by the following procedure. Note that once a child has been constructed, the roles of the two parents are reversed and a second child is obtained using the same procedure.

Every city of the mother individual is progressively inserted into the father city tour. The order in which cities are inserted is the order of the mother city tour. We determine the insertion position of a city v_{j_k} as follows: we consider every insertion position of v_{j_k} between cities v_{i_l} and $v_{i_{l+1}}$ of the father such that $W_{i'_l} \neq W_{j'_k}$ and $W_{i'_{l+1}} \neq W_{j'_k}$ (thus avoiding that two identical clusters follow); among these possibilities, the one minimizing insertion cost $c_{i_l j_k} + c_{j_k i_{l+1}} - c_{i_l i_{l+1}}$ is chosen.

Once every city of the mother individual is inserted, we derive a cluster sequence in which every cluster appears twice. This sequence is called the redundant-sequence. The next step is to determine an optimal feasible subsequence, *i.e.*, where every cluster is visited exactly once.

The search is computed through a dynamic programming algorithm, applied

to a graph obtained from the redundant-sequence. This graph is built by the following procedure. A vertex is inserted for every city of every cluster, once for every appearance of the cluster in the sequence. Basically, an arc is added for every pair of vertices issued from different cluster position in the sequence, in the direction of the sequence (see Figure 1 for an aggregated vision of the graph and Figure 2 for an extract of the real graph). Some graph reductions will however be defined subsequently (see Section 3.3). The objective is to find the shortest path in the graph between the two extreme clusters, with the constraints that every cluster must be visited exactly once and that the solution must be a cycle.

Before giving more details on the dynamic programming algorithm, let us illustrate the behavior of this crossover operator on a simple example. Consider a set of clusters $W = \{W_1, \ldots, W_5\}$. The father and mother cluster tours are:

$$\text{father: } W_4\ W_3\ W_1\ W_5\ W_2\ W_4$$
$$\text{mother: } W_4\ W_1\ W_3\ W_5\ W_2\ W_4$$

The insertion procedure defines a redundant-sequence of clusters of the form:

$$\text{child: } \underline{W_4}\ W_1\ \underline{W_3}\ \underline{W_1}\ \underline{W_5}\ W_2\ W_3\ \underline{W_2}\ W_5\ \underline{W_4}$$

where the clusters from the father node are underlined and the insertion positions are defined using the city tours of the individuals.

The graph represented by Figures 1 and 2 is then implicitly defined. From this graph, the dynamic programming algorithm determines an optimal city tour, from which the cluster sequence defining the new individual is derived. The implementation of this algorithm is detailed in Section 3.3.

The main advantage of this operator is to span a very large solution space. Indeed, the number of feasible cluster subsequences of the redundant-sequence is $O(2^m)$. Furthermore, for a given subsequence, the number of city tours is $O((n/m)^m)$ (it can be easily seen that the largest space is obtained when every cluster has the same size n/m). As a consequence, the solution defined by the crossover operator is the best among $O(2^m(n/m)^m)$ solutions. In our sense, it both allows high diversification and populations of good quality. However, of

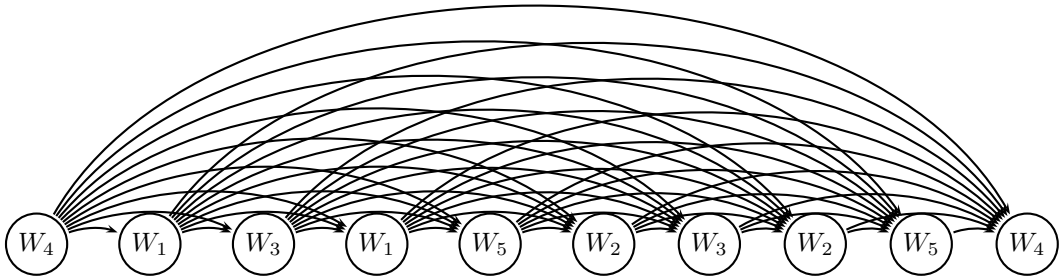

Figure 1: Graph obtained from the redundant-sequence: aggregated view of clusters

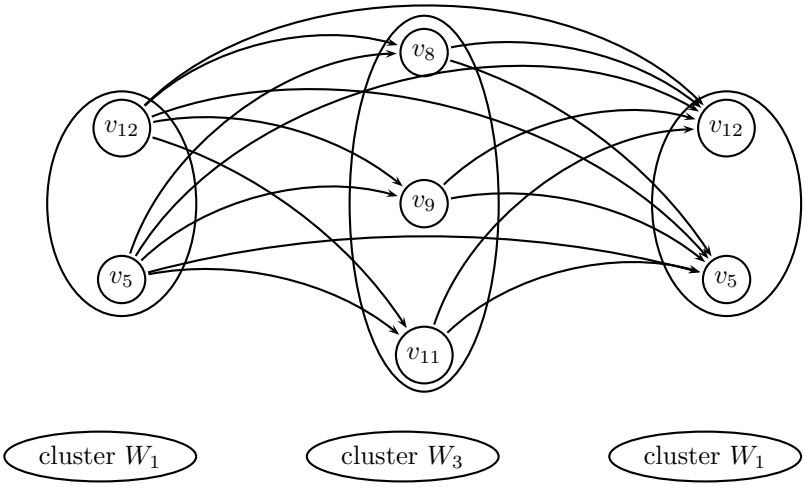

Figure 2: Graph obtained from the redundant-sequence: precise view (extract)

course, the price to pay is a high computation cost compared to classic crossover operators. The objective of this work is to evaluate whether this price is worth being paid or not.

3.3. Detailed implementaion of the crossover operator

3.3.1. Dynamic programming algorithm

The dynamic programming algorithm used to find the shortest path in the graph defined in the previous section is inspired from the algorithm developed in Feillet *et al.* [31] for the Elementary Shortest Path Problem with Resource Constraints. This algorithm is an extension of the classical Bellman's labelling algorithm [32]. Our algorithm starts with initial labels associated to every vertex of the first cluster. Labels are then extended iteratively with the constraint that every cluster has to be visited exactly once. This constraint is treated like the elementary path constraint in [31]. The graph being acyclic, extending the labels in the topological order of the vertices provides the optimal path.

In the following, we denote each label by $L = (C, \delta_1, \ldots, \delta_m)$, where C is the cost of the partial path represented by L and $\delta_i \in \{0, 1\}$ indicates whether cluster W_i is present in the path or not. The extension of a label through an arc is feasible when $\delta_i = 0$ for the cluster W_i of the destination city, except when this cluster is the last cluster of the sequence. In this case, the feasibility conditions are that the destination city is the first city of the partial path and that $\delta_i = 1$ for $1 \leq i \leq m$.

A label L^1 dominates a label L^2, which is noted $L^1 < L^2$, when the two partial paths represented by these labels lead to the same vertex and one can be sure that any extension of L^1 is going to be cheaper than the identical extension for L^2. Here, $L^1 < L^2$ when $C^1 \leq C^2$, $\delta_i^1 \geq \delta_i^2$ for $1 \leq i \leq m$ and the first and last cities of the two partial paths are identical. Under these conditions, L^2 can be deleted.

3.3.2. Lower bound

Every time a new label $L = (C, \delta_1, \ldots, \delta_m)$ is extended, a lower bound on the cost of any path that could be obtained from this label, is computed. This lower bound is compared with an upper bound initially defined as the value of the father. This upper bound is valid seeing that the city tour of the father exists in the graph. The upper bound is updated each time a new best solution

is found by the algorithm. When the lower bound is greater than the upper bound, L is deleted.

The lower bound $LB(L)$ is given by the following formula:

$$LB(L) = C + \sum_{\{1 \leq j \leq m, \delta_j = 0\}} C_{lj}$$

where l is the position in the redundant-sequence of the cluster to which L has just been extended and C_{lj} is the minimal cost incurred by the future visit of cluster W_j.

C_{lj} is computed as the minimum arc cost among arcs whose:

1. destination is one of the cities of the last occurence of W_j in the redundant-sequence,
2. origin is one of the cities located between the cluster in position l (included) and the last occurence of W_j in the redundant-sequence.

Values C_{lj} are computed in a pre-processing phase, as soon as the redundant-sequence is set, for every position l of the sequence and for every cluster W_j. The time complexity of this computation is $O(nm)$.

Note that, it might happens that the last occurence of W_j precedes position l. In this case C_{lj} is set to a large value and the label is automatically deleted if $\delta_j = 0$ since cluster W_j is unreachable.

3.3.3. Graph reduction

Since every cluster has to be visited, edges skipping all occurrences of a cluster can be removed. Moreover, two occurrences of a cluster do not need to be connected. Finally, a cluster only needs to be connected to the first occurrence of any other cluster located after him in the redundant-sequence. Figure 3 presents the graph obtained applying these rules from the graph of Figure 1.

3.3.4. Heuristic speed-ups

In order to limit the time consumed by the crossover operator, we have implemented two simple heuristic speed-ups.

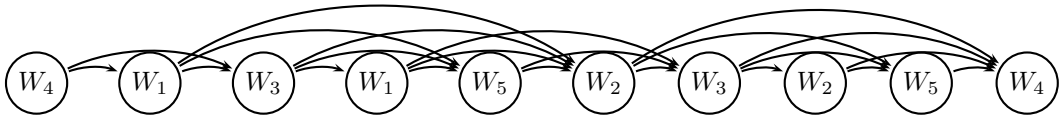

Figure 3: Reduced graph obtained from the redundant-sequence: aggregated view of clusters

Limitation of the size of the label lists. Throughout the dynamic programming algorithm, a list of labels is associated with every city. Despite the dominance rule, these lists may be significantly long. The purpose here is to limit their size. A unique limit is set (100 in our experiments). An evaluation rule is defined to determine which labels should be removed when the size of a list exceeds the limit. The labels removed are those with the greatest evaluations. The evaluation $eval(L)$ of a label $L = (C, \delta_1, \ldots, \delta_m)$ is:

$$eval(L) = C + \frac{UB_0}{\sum_{\{1 \leq j \leq m\}} C_{1j}} \sum_{\{1 \leq j \leq m, \delta_j = 0\}} C_{lj}$$

where UB_0 is the cost of the father individual and l is the position of the label in the redundant-sequence. $\sum_{\{1 \leq j \leq m\}} C_{1j}$ is the lower bound cost of an empty label (whose cost is equal to 0), $\sum_{\{1 \leq j \leq m, \delta_j = 0\}} C_{lj}$ is the lower bound of the extension of the label L towards the clusters which have not been visited yet. The formula aims to balance the actual cost of the label and an evaluation of the extension cost (as proposed in Section 3.3.2). The two terms are normalized so that the evaluations of an initial label (with a cost equal to 0) and of the label corresponding to the father city tour have an identical value UB_0.

Reduction of the clusters. The purpose here is to limit the size of the graph, by removing some cities from the different clusters. A measure is defined to

evaluate the attractiveness of a city. The less attractive cities are removed for each cluster of the redundant-sequence. The measure quantifying the interest of city v_k of the cluster at position l of the sequence is:

$$eval(k,l) = \sum_{v_i \in \Gamma^-(l)} \sum_{v_j \in \Gamma^+(l)} c_{ik} + c_{kj} - c_{ij}$$

where $\Gamma^-(l)$ (respectively, $\Gamma^+(l)$) is the set of cities belonging to the two clusters preceding (respectively, following) position l in the redundant-sequence. This measure gives a tendency on the insertion cost of city v_k in a solution. Based on this measure, the maximum size of a cluster W_i is set to $\lceil |W_i|^\rho \rceil$, where ρ is a parameter (0.8 in our experiments). Note that with this formula the percentage of cities removed from a cluster increases with its size.

3.3.5. Complexity of the dynamic programming algorithm

It is interesting to note that the algorithm described in section 3.3.1 does not achieve a polynomial time complexity. The objective of this section is to give some more insights into this complexity. In this analysis, we do not consider the two heuristic speed-ups described above.

A state is defined for every vertex of the graph and for every value of resources $\{\delta_1, \ldots, \delta_m\}$. The graph contains $2n$ cities. Resources δ_i are binary. Hence the number of states is $O(n2^m)$. The label associated with a state is extended toward a maximum of n other states. Every new label is inserted in a label list of maximal size 2^m. The insertion consists in the comparison with every label of the list. Each comparison has a complexity $O(m)$. The cost of inserting a new label in a list is then $O(m2^m)$ and the cost of extending a label $O(nm2^m)$

One can deduce that the worst case complexity of the algorithm is $O(n^2m2^{2m})$. Obviously, one can expect that the number of operations is significantly reduced in practice. Note also that with the limitation of the size of the label lists, the complexity becomes $O(n^2m)$.

3.4. Local search heuristics

The local search procedures presented here are applied in order to improve the quality of the individuals, both for the initial population and for every new child obtained from crossover. We first present the set of local search operators included in the Memetic Algorithm and then explain how they are managed.

3.4.1. 2-opt

This procedure is well-known in the context of the TSP (see [33] for more details). A *2-opt* move consists in choosing two arcs in the city tour, permuting the circulation between the ending vertices of these arcs and reconnecting the tour. The complexity of a move is $O(m^2)$ where m is the number of clusters. *2-opt* moves are repeated as long as improvements are achieved.

3.4.2. 3-opt

Similar to the *2-opt*, the *3-opt* (presented also in [33]) chooses three arcs in the tour and de-interlace the path between the ending cities of these arcs. The complexity of the procedure is $O(m^3)$. Again, moves are repeated as long as improvements are achieved.

3.4.3. Lin-Kernighan

The *Lin-Kernighan* algorithm [34] is one of the best heuristics for Euclidean Traveling Salesman Problems. Briefly, it involves swapping pairs of subtours to make a new tour. It is a generalization of *2-opt* and *3-opt* We use the implementation of this heuristic available on the Concorde website[1].

3.4.4. Move

The *Move* operator considers the cluster sequence of the individual, selects a cluster and determine the best position for this cluster in the sequence. This move is applied once for every cluster. To determine the best position for a given cluster W_i, a redundant-sequence is created where W_i is first removed and then re-inserted between every pair of clusters. The dynamic programming

[1] http://www.tsp.gatech.edu/concorde/DOC/index.html

presented in Sections 3.2 and 3.3 is then applied to recover the best feasible city tour.

The *Move* operator can be illustrated with the following example. Let W_4 W_3 W_1 W_5 W_2 W_4 be the cluster tour of an individual. When applying the operator to cluster W_1, redundant-sequence W_4 W_1 W_3 W_1 W_5 W_1 W_2 W_1 W_4 is obtained. A best city tour is then computed in the corresponding graph (see Figure 4).

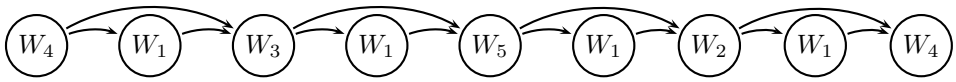

Figure 4: Graph obtained from the redundant-sequence (*Move* operator): aggregated view of clusters

The neighborhood defined by the *Move* operator has a size $O(m(n/m)^m)$ (computed as explained in Section 3.2 for the crossover operator). In the dynamic programming algorithm a label L is defined by a pair (C, δ_i). The number of states is thus $O(n)$ and the complexity of the procedure is $O(nm)$.

3.4.5. Control of the local search operators

The call to the previous operators is controled in the following fashion. When a new individual is introduced into the population during the initialization phase, *2-opt*, *3-opt* and *Lin-Kernighan* procedures are successively applied, in this order.

When a new child is computed with the crossover operator, one of the two following local search schemes is applied with a probability 0.5:

- apply *2-opt*, *3-opt* and *Move*, in this order,
- apply *Lin-Kernighan*.

4. Computational Results

The algorithm was coded in C++ and run on an Intel Pentium IV 2.00 Ghz and 2.0 GB RAM under Linux/Debian. Instances used are part of the GTSPLIB library[2] which proposes a set of 65 instances. Among these instances, we have selected 54 instances defined in [9] and used for comparison in most previous papers ([20, 18, 9, 22]). In these instances, clusters contain cities, generally 5, located in a same geographical area. The number of clusters varies between 10 and 217.

The optimal solutions is always known (provided by the Branch & Cut algorithm from Fischetti *et al.* [9]) for the selected instances with $10 \leq m \leq 89$. For the remaining instances ($99 \leq m \leq 217$), Silberholz and Golden [22] provide best known results, which in fact are the average results on 5 trials for their algorithm and Snyder and Daskin's algorithm. It is important to note that the value of the best solution found by these algorithms for these instances is not available.

For all experiments, we set the number of individuals per population (N) to 50, the number of crossovers ($2 \times k$) to 30, the maximum number of iterations (N_1) to 100 and the maximum number of iterations without improving the best solution (N_2) to 10 and the probability of mutation (μ) to 0.05 (which is the value found in [22]). The computational results presented are the mean results obtained through 5 attempts for each instance.

Table 1 evaluates the performances of our algorithm regarding the gap with optimal solutions or best known solutions. The column headings are defined as follows:

- instance: the name of the test problem; the digits at the beginning of the name give the number of clusters, those at the end give the number of nodes;
- best: the optimal objective value when known or the best known averaged

[2]GTSPLIB is available at the address http://www.cs.rhul.ac.uk/home/zvero/GTSPLIB/

solution value;

- value : the value of our algorithm for one trial;
- CPU : the CPU time in seconds for one execution of our algorithm;
- ♯opt: the number of trials, out of five, for which our algorithm found the optimal solution (when known);
- mean gap: the mean gap of our algorithm above the optimum or the best known averaged solution (percentage);
- min gap: the minimal gap of our algorithm above the optimum or the best known averaged solution (percentage);
- max gap: the maximal gap of our algorithm above the optimum or the best known averaged solution (percentage);
- CPU time: the mean CPU time in seconds for one execution of the algorithm.

Results written in bold represent cases for which the solution we found is equal to the optimal one or improve the best known averaged solution.

Table 1: Experimental results: quality of the solutions

instance	best	value	CPU	♯opt	mean gap	min gap	max gap	CPU
10att48.gtsp	5394	**5394**	0.57	5	**0.00**	**0.00**	**0.00**	0.76
10gr48.gtsp	1834	**1834**	0.97	5	**0.00**	**0.00**	**0.00**	0.79
10hk48.gtsp	6386	**6386**	0.58	5	**0.00**	**0.00**	**0.00**	0.50
11eil51.gtsp	174	**174**	0.84	5	**0.00**	**0.00**	**0.00**	0.81
12brazil58.gtsp	15332	**15332**	0.85	5	**0.00**	**0.00**	**0.00**	0.65
14st70.gtsp	316	**316**	1.02	5	**0.00**	**0.00**	**0.00**	0.93
16eil76.gtsp	209	**209**	1.18	5	**0.00**	**0.00**	**0.00**	1.00
16pr76.gtsp	64925	**64925**	1.27	5	**0.00**	**0.00**	**0.00**	1.17
20kroA100.gtsp	9711	**9711**	1.98	5	**0.00**	**0.00**	**0.00**	1.81
20kroB100.gtsp	10328	**10328**	2.01	5	**0.00**	**0.00**	**0.00**	2.17
20kroC100.gtsp	9554	**9554**	1.84	5	**0.00**	**0.00**	**0.00**	1.85
20kroD100.gtsp	9450	**9450**	2.93	5	**0.00**	**0.00**	**0.00**	2.77
20kroE100.gtsp	9523	**9523**	1.87	5	**0.00**	**0.00**	**0.00**	1.81
20rat99.gtsp	497	**497**	3.52	5	**0.00**	**0.00**	**0.00**	3.89
20rd100.gtsp	3650	**3650**	2.93	5	**0.00**	**0.00**	**0.00**	2.91
21eil101.gtsp	249	**249**	2.07	5	**0.00**	**0.00**	**0.00**	2.09
21lin105.gtsp	8213	**8213**	3.25	5	**0.00**	**0.00**	**0.00**	3.18
22pr107.gtsp	27898	**27898**	4.67	5	**0.00**	**0.00**	**0.00**	4.78
24gr120.gtsp	2769	**2769**	2.30	5	**0.00**	**0.00**	**0.00**	2.34
25pr124.gtsp	36605	**36605**	2.89	5	**0.00**	**0.00**	**0.00**	2.84
26bier127.gtsp	72418	**72418**	3.02	5	**0.00**	**0.00**	**0.00**	3.35
28pr136.gtsp	42570	**42570**	4.17	5	**0.00**	**0.00**	**0.00**	4.23

Table 1: Experimental results: quality of the solutions

instance	best	value	CPU	♯opt	mean gap	min gap	max gap	CPU
29pr144.gtsp	45886	**45886**	5.38	5	**0.00**	**0.00**	**0.00**	5.42
30kroA150.gtsp	11018	**11018**	5.27	5	**0.00**	**0.00**	**0.00**	5.95
30kroB150.gtsp	12196	**12196**	4.61	5	**0.00**	**0.00**	**0.00**	5.02
31pr152.gtsp	51576	**51576**	4.45	5	**0.00**	**0.00**	**0.00**	5.24
32u159.gtsp	22664	**22664**	5.53	5	**0.00**	**0.00**	**0.00**	5.58
39rat195.gtsp	854	**854**	10.42	5	**0.00**	**0.00**	**0.00**	11.01
40d198.gtsp	10557	**10557**	8.72	5	**0.00**	**0.00**	**0.00**	10.15
40kroA200.gtsp	13406	**13406**	6.74	5	**0.00**	**0.00**	**0.00**	10.41
40kroB200.gtsp	13111	**13111**	8.78	5	**0.00**	**0.00**	**0.00**	10.81
45ts225.gtsp	68340	**68340**	35.31	3	0.04	**0.00**	0.09	31.45
46pr226.gtsp	64007	**64007**	6.92	5	**0.00**	**0.00**	**0.00**	8.25
53gil262.gtsp	1013	**1013**	25.12	2	0.14	**0.00**	0.3	24.34
53pr264.gtsp	29549	**29549**	16.64	5	**0.00**	**0.00**	**0.00**	18.27
60pr299.gtsp	22615	**22615**	20.19	5	**0.00**	**0.00**	**0.00**	21.25
64lin318.gtsp	20765	**20765**	24.89	5	**0.00**	**0.00**	**0.00**	26.33
80rd400.gtsp	6361	**6361**	38.33	1	0.42	**0.00**	0.75	32.21
84fl417.gtsp	9651	**9651**	21.9	5	**0.00**	**0.00**	**0.00**	31.63
88pr439.gtsp	60099	**60099**	56.46	5	**0.00**	**0.00**	**0.00**	42.55
89pcb442.gtsp	21657	21673	76.64	1	0.19	**0.00**	0.38	62.53
99d493.gtsp	20117.2	**20073**	115.08		**-0.03**	**-0.28**	0.23	166.11
107att532.gtsp	13510.8	**13470**	67.08		**-0.30**	**-0.34**	**-0.17**	137.54
107si535.gtsp	13513.2	**13502**	69.49		**-0.01**	**-0.08**	0.06	90.98
113pa561.gtsp	1051.2	**1046**	165.64		**-0.84**	**-1.26**	**-0.21**	149.43
115rat575.gtsp	2414.8	**2408**	155.97		0.04	**-0.45**	0.09	157.01
131p654.gtsp	27439	**27428**	74.81		**-0.03**	**-0.04**	**0.00**	144.95
132d657.gtsp	22599	**22533**	234.68		**-0.15**	**-0.43**	0.28	259.11
145u724.gtsp	17370.6	17448	214.98		0.45	0.24	0.66	218.66
157rat783.gtsp	3300.2	**3290**	462.74		**-0.07**	**-0.58**	0.12	391.79
201pr1002.gtsp	114582.2	114751	415.69		0.03	**-0.18**	0.16	513.48
207si1032.gtsp	22388.8	**22348**	680.95		**-0.26**	**-0.33**	**-0.17**	616.28
212u1060.gtsp	108390.4	**107395**	663.44		**-0.92**	**-1.58**	0.19	762.86
217vm1084.gtsp	131884.6	**131345**	613.76		**-0.26**	**-0.65**	0.09	583.44

Table 1 shows that, with 5 attempts, 41 instances out of 41 are optimally solved and that for 37 of these instances, the optimal solution is found at each run of the Memetic Algorithm. The difference between the best and the worst solution returned from the 5 trials always remains small (never exceeding 0.75%), which tends to indicate that our algorithm is robust.

For larger instances, the table shows that the mean results of our algorithm outperform the best known averaged solutions for 10 instances out of 13. For 3 instances, solutions are even improved by our worst result. In the worst case, the results produced by our algorithm never exceeds a gap equal to 0.66% with the best known averaged results, which confirms the robustness of the algorithm.

Table 2 presents our best results within five trials for twelve instances for which the best known averaged solution value is improved.

Table 2: Experimental results: Best solutions found

instance	best	this paper
99d493.gtsp	20117.2	20061
107att532.gtsp	13510.8	13464
107si535.gtsp	13513.2	13502
113pa561.gtsp	1051.2	1038
115rat575.gtsp	2414.8	2404
131p654.gtsp	27439	27428
132d657.gtsp	22599	22502
157rat783.gtsp	3300.2	3281
201pr1002.gtsp	114582.2	114374
207si1032.gtsp	22388.8	22315
212u1060.gtsp	108390.4	106677
217vm1084.gtsp	131884.6	131028

In order to measure more precisely the efficiency of the crossover procedure proposed in this paper, we have also implemented a simpler and more classic crossover procedure, namely the one-point crossover (Goldberg [35]). The parameters are left unchanged for both crossover procedures. Table 3 compares these crossovers and the results of our algorithm without the mutation operator. In this table, the crossover based on large neighborhood search is called LNS crossover and our algorithm without the mutation operator "LNS w/o Mutation". The results presented are the mean results obtained through 5 attempts for each instance. The column headings are as follows:

- instance: the name of the test problem;
- best: the optimal objective value when known or the best known averaged solution value;
- gap: the mean gap of the algorithm above the optimal solution or the best known averaged solution (percentage);
- CPU: the CPU time in seconds.

Table 3: Experimental results : comparisons between our algorithms

		LNS		One-Point		LNS w/o Mutation	
instance	best	gap	CPU	gap	CPU	gap	CPU
10att48.gtsp	5394	**0.00**	0.76	**0.00**	0.15	**0.00**	0.94
10gr48.gtsp	1834	**0.00**	0.79	**0.00**	0.14	**0.00**	0.84
10hk48.gtsp	6386	**0.00**	0.5	**0.00**	0.17	**0.00**	0.49
11eil51.gtsp	174	**0.00**	0.81	**0.00**	0.15	**0.00**	0.73

Table 3: Experimental results : comparisons between our algorithms

		LNS		One-Point		LNS w/o Mutation	
name	best	gap	CPU	gap	CPU	gap	CPU
12brazil58.gtsp	15332	**0.00**	0.65	**0.00**	0.24	**0.00**	0.52
14st70.gtsp	316	**0.00**	0.93	**0.00**	0.2	**0.00**	0.75
16eil76.gtsp	209	**0.00**	1	**0.00**	0.17	**0.00**	0.96
16pr76.gtsp	64925	**0.00**	1.17	**0.00**	0.25	**0.00**	0.82
20kroA100.gtsp	9711	**0.00**	1.81	**0.00**	0.27	**0.00**	1.65
20kroB100.gtsp	10328	**0.00**	2.17	**0.00**	0.27	**0.00**	2.17
20kroC100.gtsp	9554	**0.00**	1.85	**0.00**	0.27	**0.00**	1.7
20kroD100.gtsp	9450	**0.00**	2.77	**0.00**	0.27	**0.00**	2.63
20kroE100.gtsp	9523	**0.00**	1.81	**0.00**	0.53	**0.00**	1.79
20rat99.gtsp	497	**0.00**	3.89	**0.00**	0.44	**0.00**	4.21
20rd100.gtsp	3650	**0.00**	2.91	0.36	0.4	**0.00**	2.61
21eil101.gtsp	249	**0.00**	2.09	0.56	0.61	**0.00**	2.23
21lin105.gtsp	8213	**0.00**	3.18	**0.00**	0.36	**0.00**	2.75
22pr107.gtsp	27898	**0.00**	4.78	0.08	0.33	**0.00**	4.65
24gr120.gtsp	2769	**0.00**	2.34	0.62	0.07	**0.00**	2.58
25pr124.gtsp	36605	**0.00**	2.84	**0.00**	0.44	**0.00**	2.47
26bier127.gtsp	72418	**0.00**	3.35	**0.00**	0.42	**0.00**	2.52
28pr136.gtsp	42570	**0.00**	4.23	0.72	0.86	**0.00**	5.61
29pr144.gtsp	45886	**0.00**	5.42	**0.00**	0.54	**0.00**	5.54
30kroA150.gtsp	11018	**0.00**	5.95	0.01	1.14	**0.00**	6.2
30kroB150.gtsp	12196	**0.00**	5.02	0.33	1.45	**0.00**	4.56
31pr152.gtsp	51576	**0.00**	5.24	**0.00**	0.68	**0.00**	3.36
32u159.gtsp	22664	**0.00**	5.58	0.43	0.83	**0.00**	6.1
39rat195.gtsp	854	**0.00**	11.01	1.05	1.63	**0.00**	10.36
40d198.gtsp	10557	**0.00**	10.15	0.07	1.41	0.17	8.4
40kroA200.gtsp	13406	**0.00**	10.41	0.21	1.65	**0.00**	7.7
40kroB200.gtsp	13111	**0.00**	10.81	0.15	2.09	**0.00**	9.75
45ts225.gtsp	68340	0.04	31.45	0.29	1.91	**0.00**	33.42
46pr226.gtsp	64007	**0.00**	8.25	**0.00**	1.03	**0.00**	7.15
53gil262.gtsp	1013	0.14	26.34	1.8	2.35	0.89	16.44
53pr264.gtsp	29549	**0.00**	18.27	0.46	2.43	**0.00**	18
60pr299.gtsp	22615	**0.00**	21.25	0.2	5.79	**0.00**	27.84
64lin318.gtsp	20765	**0.00**	26.33	0.59	4.67	3.33	31.67
80rd400.gtsp	6361	0.42	32.21	1.45	10.12	1.56	36.45
84fl417.gtsp	9651	**0.00**	31.63	**0.00**	3.41	**0.00**	29.75
88pr439.gtsp	60099	**0.00**	42.65	0.09	10.56	0.2	33.87
89pcb442.gtsp	21657	0.19	62.53	1.26	11.22	0.38	108.9
99d493.gtsp	20117.2	**-0.03**	166.11	0.03	12.1	0.82	122.73
107att532.gtsp	13510.8	**-0.30**	121.54	0.21	18.73	**-0.01**	91.29
107si535.gtsp	13513.2	**-0.01**	90.98	0.01	10.58	0.07	103.33
113pa561.gtsp	1051.2	**-0.84**	149.43	1.58	11.66	0.27	131.53
115rat575.gtsp	2414.8	0.04	157.01	5.01	17.99	2.78	151.88
131p654.gtsp	27439	**-0.03**	144.95	**-0.01**	10.95	**-0.03**	91.52
132d657.gtsp	22599	**-0.15**	259.11	1.15	27.2	2.66	203.8
145u724.gtsp	17370	0.45	218.66	2.76	45.31	1.08	241.83
157rat783.gtsp	3300.2	**-0.07**	391.79	3.68	42.76	1.08	527.74
201pr1002.gtsp	114582.2	0.03	513.48	1.49	76.54	1.67	345.26
207si1032.gtsp	22388.8	**-0.26**	616.28	0.33	75	**-0.04**	619.99
212u1060.gtsp	108390.4	**-0.92**	762.86	**-0.28**	88.6	0.78	697.36
217vm1084.gtsp	131884.6	**-0.26**	583.44	0.3	89.87	1.16	462.46

The LNS crossover shows a significant advantage in solution quality over the one-point crossover. For the smaller instances (for which the optimal solution is known), the average gap of the one-point crossover is equal to 0.26%, whereas the average gap is reduced to 0.02% with the LNS crossover. The runtimes of

the algorithms using the LNS crossover and the one-point crossover are however significantly different, the one-point crossover being six times faster.

For larger instances, the LNS crossover produces much better solutions than the one-point crossover, keeping the same differences between runtimes. The mutation operator permits to avoid premature convergence.

Table 4 finally compare our algorithm LNS (for one trial) with the mean results of our algorithm LNS (through 5 attempts), the Genetic Algorithm *mrOX* proposed by Silberholz and Golden [22], the Memetic Algorithm from Snyder and Daskin[20], the GI^3 algorithm of Renaud *et al.* [18] and the Branch & Cut algorithm from Fischetti *et al.* [9]. These comparisons are only given for the instances where the optimal solution is known.

The results have been obtained on the following computers:

- mrOX and Snyder and Daskin: Pentium IV 3.0 GHz processor and 1 GB RAM.
- GI^3: Sun Sparc Station LX.
- B& C.: HP 9000/720.

For each algorithm, two columns are presented in the table:

- gap: the mean gap of the algorithm above the optimal solution (percentage);
- CPU: the CPU time in seconds.

The average behavior of the different algorithms is given at the end of the table.

Table 4: Comparison with several algorithms

	LNS - 1 trial		LNS - 5 trials		mrOX		Snyder		GI		BC
instance	value	CPU	gap	CPU	gap	CPU	gap	CPU	gap	CPU	CPU
10att48	5394	0.57	0	0.76	0	0.36	0	0.18	*	*	2.1
10gr48	1834	0.97	0	0.79	0	0.32	0	0.08	*	*	1.9
10hk48	6386	0.58	0	0.5	0	0.31	0	0.08	*	*	3.8
11eil51	174	0.84	0	0.81	0	0.26	0	0.08	0	0.3	2.9
12brazil58	15332	0.85	0	0.65	0	0.78	0	0.1	*	*	3
14st70	316	1.02	0	0.93	0	0.35	0	0.07	0	1.7	7.3
16eil76	209	1.18	0	1	0	0.37	0	0.11	0	2.2	9.4
16pr76	64925	1.27	0	1.17	0	0.45	0	0.16	0	2.5	12.9
20kroA100	9711	1.98	0	1.81	0	0.5	0	0.24	0	5	51.5

Table 4: Results for several algorithms

	LNS - 1 trial		LNS - 5 trials		mrOX		Snyder		GI		BC
instance	value	CPU	gap	CPU	gap	CPU	gap	CPU	gap	CPU	CPU
20kroB100	10328	2.01	0	2.17	0	0.63	0	0.25	0	6.8	18.4
20kroC100	9554	1.84	0	1.85	0	0.6	0	0.22	0	6.4	22.2
20kroD100	9450	2.93	0	2.77	0	0.62	0	0.23	0	6.5	14.4
20kroE100	9523	1.87	0	1.81	0	0.67	0	0.43	0	8.6	14.3
20rat99	497	3.52	0	3.89	0	0.58	0	0.15	0	6.7	13
20rd100	3650	2.93	0	2.91	0	0.51	0	0.29	0.08	7.3	16.6
21eil101	249	2.07	0	2.09	0	0.48	0	0.18	0.4	5.2	25.6
21lin105	8213	3.25	0	3.18	0	0.6	0	0.33	0	14.4	16.4
22pr107	27898	4.67	0	4.78	0	0.53	0	0.2	0	8.7	7.4
24gr120	2769	2.30	0	2.34	0	0.66	0	0.32	*	*	41.9
25pr124	36605	2.89	0	2.84	0	0.68	0	0.26	0.43	12.2	25.9
26bier127	72418	3.02	0	3.35	0	0.78	0	0.28	5.55	36.1	23.6
28pr136	42570	4.17	0	4.23	0	0.79	0.16	0.36	1.28	12.5	43
29pr144	45886	5.38	0	5.42	0	1	0	0.44	0	16.3	8.2
30kroA150	11018	5.27	0	5.95	0	0.98	0	0.32	0	17.8	100.3
30kroB150	12196	4.61	0	5.02	0	0.98	0	0.71	0	14.2	60.6
31pr152	51576	4.45	0	5.24	0	0.97	0	0.38	0.47	17.6	94.8
32u159	22664	5.53	0	5.58	0	0.98	0	0.55	2.6	18.5	146.4
39rat195	854	10.42	0	11.01	0	1.37	0	1.33	0	37.2	245.9
40d198	10557	8.72	0	10.15	0	1.63	0.07	1.47	0.6	60.4	763.1
40kroA200	13406	6.74	0	10.41	0	1.66	0	0.95	0	29.7	187.4
40kroB200	13111	8.78	0	10.81	0.05	1.63	0.01	1.29	0	35.8	268.5
45ts225	68340	35.31	0.04	31.45	0.14	1.71	0.28	1.09	0.61	89	37875.9
46pr226	64007	6.92	0	8.25	0	1.54	0	1.09	0	25.5	106.9
53gil262	1013	25.12	0.14	24.34	0.45	3.64	0.55	3.05	5.03	115.4	6624.1
53pr264	29549	16.64	0	18.27	0	2.36	0.09	2.72	0.36	64.4	337
60pr299	22615	20.19	0	21.25	0.05	4.59	0.16	4.08	2.23	90.3	812.8
64lin318	20765	24.89	0	26.33	0	8.08	0.54	5.39	4.59	206.8	1671.9
80rd400	6361	38.33	0.42	32.21	0.58	14.58	0.72	10.27	1.23	403.5	7021.4
84fl417	9651	21.9	0	31.63	0.04	8.15	0.06	6.18	0.48	427.1	16719.4
88pr439	60099	56.46	0	42.55	0	19.06	0.83	15.09	3.52	611	5422.8
89pcb442	21673	76.64	0.19	62.53	0.01	23.43	1.23	11.74	5.91	567.7	58770.5
Averages	0.001	9.12	0.02	10.12	0.03	2.69	0.11	1.77	0.98	83.09	3356.47
Trials	1		5		5		5		1		1

Table 4 shows that our algorithm produces in average the better solution, compared to the other heuristics. The average gap with optimal solutions is only 0.02%.

Runtime comparisons with other heuristics are difficult because different computers with various computing powers were used. The table however shows that our algorithm is much slower than mrOX algorithm (which runs on a faster computer), for a limited improvement in terms of quality, or the Snyder and Daskin's algorithm, for a more significant improvement. Note however that the presence of many easy instances solved optimally by all algorithms tends to tighten the gaps.

5. Conclusions

In this paper, we proposed to solve the GTSP using a Memetic Algorithm where the crossover operator relies on large neighborhood search. Our main contribution is the originality of our crossover procedure. Experimental results show that our algorithm is robust and presents a good balance between CPU time and quality of the solutions. 41 out of the 41 problems for which the optimal solution is known are solved optimally. 10 out of 13 best known averaged solution values are improved. Among all the executions of the algorithm the worst solution returned exhibits a gap of 0.66 % with the best known solution.

References

[1] P. Moscato, C. Cotta, A Gentle Introduction to Memetic Algorithms, Operations Research & Management Science 57 (2) (2005) 105–144.

[2] W. E. Hart, N. Krasnogor, J. E. Smith, Recent Advances in Memetic Algorithms, Springer, 2005.

[3] R. K. Ahuja, O. Ergun, J. B. Orlin, A. Punnen, A survey of very large-scale neighborhood search techniques, Discrete Applied Mathematics 123 (2002) 75–102.

[4] G. Reinelt, TSPLIB - A Traveling Salesman Problem Library, ORSA Journal on Computing 3 (1991) 376–384.

[5] S. S. Srivastava, S. Kumar, R. C. Garg, P. Sen, Generalized Traveling Salesman Problem through n sets of nodes, CORS Journal 7 (1969) 97–101.

[6] A. L. Henry-Labordere, The record balancing problem : A dynamic programming solution of a generalized traveling salesman problem, RAIRO B2 (1969) 43–49.

[7] G. Laporte, Y. Nobert, Generalized Travelling Salesman Problem through n Sets of Nodes : An integer programming approach, INFOR 31 (1984) 61–75.

[8] G. Laporte, H. Mercure, Y. Nobert, Generalized Travelling Salesman Problem through n Sets of Nodes : the Asymmetrical Case, Discrete Applied Mathematics 18 (1984) 185–197.

[9] M. Fischetti, J. J. Salazar-González, P. Toth, A Branch-and-Cut algorithm for the Symmetric Generalized Traveling Salesman Problem, Operations Research 45 (3).

[10] Y.-N. Lien, E. Ma, B. Wah, Transformation of the Generalized Traveling-Salesman Problem into the Standard Traveling-Salesman Problem, Information Sciences 74 (1993) 177–189.

[11] C. E. Noon, J. C. Bean, An Efficient Transformation of the Generalized Traveling Salesman Problem, INFOR 31 (1993) 39–44.

[12] V. Dimitrijevic, Z. Saric, An Efficient Transformation of the Generalized Traveling Salesman Problem into the Traveling Salesman Problem on Digraphs, Information Sciences 102 (1997) 105–110.

[13] G. Laporte, F. Semet, Computational evaluation of a transformation procedure for the symmetric generalized traveling salesman problem, INFOR 37 (1999) 114–120.

[14] D. Ben-Arieh, G. Gutin, M. Penn, A. Yeo, A. Zverovitch, Transformations of generalized ATSP into ATSP, Operations Research Letters 31 (2003) 357–365.

[15] P. Slavík, On the approximation of the generalized traveling salesman problem, Tech. rep., Department of Computer Science, SUNY-Buffalo (1997).

[16] N. Garg, G. Konjevod, A Polylogarithmic Approximation Algorithm for the Group Steiner Tree Problem, Journal of Algorithms 37 (2000) 66–84.

[17] C. E. Noon, J. C. Bean, A Lagrangian Based Approach for the Asymmetric Generalized Traveling Salesman Problem, Operations Research 39 (1991) 623–632.

[18] J. Renaud, F. F. Boctor, An Efficient Composite Heuristic for the Symmetric Generalized Traveling Salesman Problem, European Journal of Operational Research 108 (1998) 571–584.

[19] J. Renaud, F. F. Boctor, G. Laporte, A fast composite heuristic for the symmetric traveling salesman problem, INFORMS Journal on Computing 8 (1996) 134–143.

[20] L. V. Snyder, M. S. Daskin, A Random-Key Genetic Algorithm for the Generalized Traveling Salesman Problem, European Journal of Operational Research 174 (2006) 38–53.

[21] X. H. Shi, Y. C. Liang, H. P. Lee, C. Lu, Q. X. Wang, Particle swarm optimization-based algorithms for TSP and generalized TSP, Information Processing Letters 103 (2007) 169–176.

[22] J. Silberholz, B. Golden, The Generalized Traveling Salesman Problem: A New Genetic Algorithm Approach, 2007, pp. 165–181.

[23] M. D. Vose, The Simple Genetic Algorithm: Foundations and Theory, MIT Press, 1998.

[24] K. F. Man, K. S. Tang, S. Kwong, Genetic Algorithms: Concepts and Designs, Springer, 1999.

[25] G. Winter, J. Périaux, M. Galaán, P. Cuesta, Genetic Algorithms in Engineering and Computer Science, Wiley, 1995.

[26] P. Moscato, M. G. Norman, A Memtic approach for the traveling salesman problem implementation of a computational ecology for combinatorial optimization on message-passing systems, in: Parallel Computing and Transputer Applications, IOS Press, Amsterdam, 1992, pp. 177–186.

[27] H. Mühlenbein, M. G. Schleuter, O. Krämer, Evolution algorithms in combinatorial optimization, Parallel Computing 7 (1988) 65–85.

[28] H. K. Tsai, J. M. Yang, Y. F. Tsai, C. Y. Kao, Some issues of designing genetic algorithms for traveling salesman problems, Soft Computing 8 (2004) 689–697.

[29] B. Bontoux, D. Feillet, Ant Colony Optimization for the Traveling Purchaser Problem, Computers & Operations Research 35 (2008) 628–637.

[30] C. Prins, A simple and effective evolutionary algorithm for the VRP, Computers & Operations Research 31 (2004) 1985–2002.

[31] D. Feillet, P. Dejax, M. Gendreau, C. Gueguen, An exact algorithm for the Elementary Shortest Path Problem with Resource Constraints: application to some vehicle routing problems, Networks 44 (2004) 216–229.

[32] R. Bellman, Dynamic programming, Princeton, New Jersey: Princeton University Press. XXV, 1957.

[33] S. Lin, Computer solutions of the traveling salesman problem 44 (1965) 2245–2269.

[34] S. Lin, B. W. Kernighan, An Effective Heuristic Algorithm for the Traveling-Salesman Problem, Operations Research 21 (1973) 498–516.

[35] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Longman Publishing Co., Inc., 1989.